Intermediate Level Components for Reconfigurable Platforms

Erwan Fabiani, Christophe Gouyen, and Bernard Pottier

Architectures et Systèmes*,
Université de Bretagne Occidentale, Brest, France

Abstract. Development productivity is a central point for the acceptance of reconfigurable platforms. Due to the availability of generic low level tools and powerful logic synthesis tools, it becomes possible to define portable components that have both a high level behavior and attributes for physical synthesis. The behavior of a component can be fixed at compile time using concise specifications that will reduce the cost and delays in developments. The method allowing to produce components is illustrated with two case studies.

1 Introduction

We are considering a new generation of general purpose circuits allowing to produce applications by field programming or configuration. Following FPGAs, the economic challenge of these circuits is to complement ASIC for markets where the production volume does not balance the cost of a specific SOC design, and where a quick application availability is critical. A consequence of cost and time-to-market constraints is the need to define software production methods with emphasis on designer productivity.

1.1 Scope

Among the different architectural options appearing, or likely to appear, we choose a generic architecture shown figure 1.a with the following parts: (1) a dedicated system processor (SP) in charge of tasks and circuit management, (2) a network on chip possibly simple and controlled by SP, (3) several heterogeneous compute units (CU) such as processors, reconfigurable data path or fine grain banks, These units have their own local memories for data, and code or configurations, (4) a memory cache, (5) several input/output units with, possibly, specific support outside the circuit.

There are two main motivations for the choice of such a distributed architecture. One is scalability, with the need to have an evolving choice of off-the-shelf circuits adapted to different kind of applications. A permanent problem with current FPGA technology is the change of scale and the actual difficulty to implement system level communications in an efficient way. The use of a network

* `http://as.univ-brest.fr`

A. Pimentel and S. Vassiliadis (Eds.): SAMOS 2004, LNCS 3133, pp. 59–68, 2004.

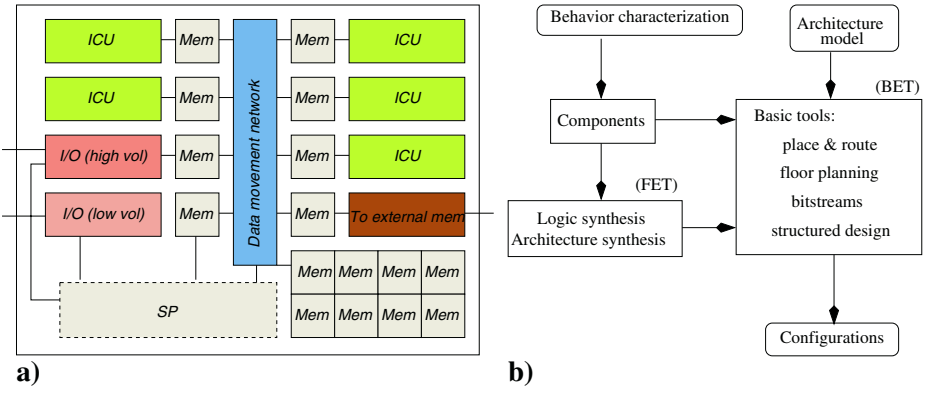

Fig. 1. (a) A parallel heterogeneous reconfigurable platform. (b) Position of the component layer related to synthesis tools and basic tools.

on chip allows to merge SOC IPs within the platform, and to control congestion in the routing resources during system activities. The other motivation is heterogeneity, meaning that it will be possible to select a set of reconfigurable or programmable resources suiting the application needs. This implies the development of tools allowing to produce different code or configurations from a single specification, depending on the architecture and execution constraints.

The platform must support compute intensive processes as found in stream, image and signal processing. These processes will use on chip input/output facilities, buffers, memory buffers, and they can spread over several compute units. In this case they need to be prepared as small tasks exchanging data buffers or transactions. Intensive computation tasks will be mapped to reconfigurable units. Other processes needing specific hardware support are controllers having short reaction delays. The operating system decides resource allocation, scheduling, swaps and memory transfers.

During these last years, our research activity has been concentrated on building portable tools for reconfigurable architectures. The MADEO framework is organized in three parts, with the general flow shown figure 1.b.

The lower layer back-end (BET) proposes tools for reconfigurable architecture modeling. Several fine grain FPGA architectures have been described successfully, including commercial circuits. The models are represented using a grammar that enables a set of generic tools to produce the basic functionalities: placing cells on FPGAs, global or point to point routing, floor-planning, regular circuit design[1]. An important property of this framework is its openness allowing synthesis algorithms to build layout of application component under programmer control. Fine control on the geometry and location of components is critical for resource management, as needed in operating systems.

Above these tools, there is a front-end support (FET) for logic synthesis. The second layer uses high level object oriented specifications and produces hierarchical application components for the first layer. The basic flow is based on directed

acyclic graph of nodes representing procedure calls that will be translated into look-up tables (LUT) or call of other graphs. The second layer tools can work competitively compared to handwritten hardware implementation because data specifications are required to be richer than usual types. Our data types are based on set of values and intervals. They are automatically produced for each function in the program and propagated downward the hierarchical graph. After type inference, synthesis tools have an exact knowledge of the computation context, and are able to lead very efficient optimizations. These optimizations take place at a symbolic level, by collapsing and simplifying nodes in the LUT graph, and at the encoding level, by exchanging data for indexes, and finally at the logic level using logic synthesis algorithms[2]. Physical mapping for fine grain FPGAs have been described in [3]. Extension for reconfigurable data path code production is currently being investigated from the same set of tools as the type system also provide support for interval description.

Above these levels we are now interested to develop architecture *components* in different ways. This paper will discuss the component status, especially their important position in the design flow and the relation with the physical target.

1.2 Component Definition

Components are intermediate in the application design flow. They can be used to define the frontier from software to hardware in a transparent way for application developers. Their main characteristics are described as follow.

Modularity and reuse: Components provide a modular behavioral interface usable during application development, either directly or from a compiler. They have an object status grouping a behavioral interface, physical synthesis capabilities, rules for use, and code or configuration to be handled by the operating system. Components provide *software re-usability*, in a way similar as IP modules do in the case of SOC. They are executable in the software development environment.

Programmability or characterization: The component behavior can come from a program expressed in a domain specific languages (DSL), or there can be a fixed parametrized behavior. In each case, components carry an implicit execution architecture that will be produced at the physical level.

Software macros as used in the FPGAs environments are components of small complexity whose definitions are hidden to the programmers.

Physical synthesis: Components embed algorithms producing a physical description of the application architecture related to a reconfigurable unit target. These algorithms use building blocks in the form of other components, or specific placed and routed primitives. They compute the respective layout of these blocks, and they produce the low level interconnections. Physical synthesis algorithms are portable at least for fine grain architectures.

Support for compilers: Some components are explicit structured descriptions of hardware. It is the case for arithmetic operators, regular processing networks,

controllers. There are also components representing the necessary transformations enabling a compiler to produce circuits in a restricted context (computation graph, regular networks to be mapped on CUs).

2 Productivity Is Concise Specifications

As development productivity is becoming a serious challenge for embedded applications, it is interesting to observe how software has solved this difficulty by the past, then in which way reconfigurable architectures could help in speeding up the development process. An important factor in development productivity is the level of abstraction in which solution specifications are produced. The first benefit of abstraction is the simplicity of the expression obtained due to the meanings of the formalism. Simplicity means speed and security of solution expression, ease to develop and maintain translation and verification tools. Programming languages have achieved a gradual progress in terms of abstraction level and in terms of modularity and reuse.

Software productivity can be evaluated on metrics such as *line of codes*, or may be more accurately on *source statements* implementing equivalent functionalities. According to industrial expert sources, productivity can scale from one to ten for general purpose languages, and it is not necessary to insist on the power of expression of specific languages.

There exists at least two clear demonstrations of the interest of abstraction rising given by *virtual machines* for general purpose languages, and *domain specific languages*. In each case the basic support is provided by a particular virtual architecture or software supports that provide a fixed higher semantic level. The compiler design is usually simple due to the service offered by the underlying support and the language can be ported to different platforms by adapting this support.

Virtual Machines can be implemented as a pure software interpreter, or more efficiently in a processor micro-code. Additional supports are required for memory management and OS level primitives. Known examples are Pascal, Smalltalk, or the Transputer[4].

Domain Specific Languages(DSL) are largely used in our current software tools, producing the desirable level of abstraction related to a particular domain[5]. Examples of DSL are text processing tools (sed, awk,...), compilation tools (lex, yacc, ...), or more specific domain tools for signal processing, graphics, etc... Due to their capability to produce the application architectures, DSL can be implemented on reconfigurable platforms. Drawbacks in using DSL include the excessive specialization of data. Testing can be an issue if the formalism does not support associated execution mechanisms, and finally DSL abstraction does not avoid domain expertise from their programmers but just eases the task.

DSL and Virtual machines are expected to help considerably application module production and control. The last part of the paper will demonstrate the interest of DSL in the case of cellular automata and regular computations.

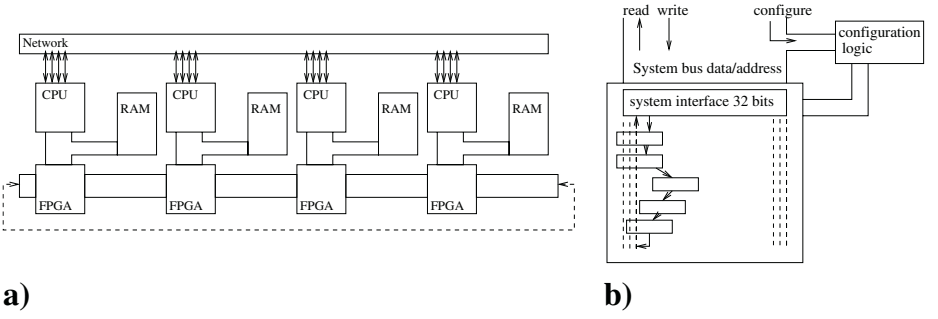

Fig. 2. Armen computer: (a)Four interconnected nodes, and (b) physical representation of computation pipelines inside the FPGA.

3 Component Design Method

The general approach for component design is bottom-up: (1) Fix the functionality to be addressed by defining what will be explicit in the parameters or program, and what will be implicit, (2) define the internal execution model, (3) define the language, (4) define the synthesis mechanisms related to a support architecture. To help the explanations, this approach will be illustrated on the example of cellular automata on the ArMen computer.

Platform description: In this case, the computer is ArMen, a distributed memory architecture whose nodes are fully interconnected using serial links (figure 2.a). Each node processor has an attached FPGA accessed in locked step read or write transactions. The processor has also support in its address space to write and read configurations to the FPGA (figure 2.b). FPGAs are connected together and can exchange data asynchronously, there is no global clock. The interface from the local system bus to the FPGA is fixed and generally used to feed a pipeline. Inside the FPGA, physical synthesis generally proceeds by allocating logic resources along a pipeline using local routes. The pipeline stages are connected to long lines by three-state buffers. Long lines is an internal bus that can bring back results to the interface with short delays. The vertical pipeline advances under processor control while other computations usually take place horizontally in an asynchronous way[6].

Functionality: Cellular automata (CA) is a well known paradigm where a discrete space of cells progresses synchronously. To define a CA it is needed to fix: (1) a neighborhood representing the dependencies relative to a cell, (2) a transition function describing the evolution of a cell given its current state and the neighbor states, (3) the geometry of the cell space and its initial value.

A CA specification must *explicit* these three points letting the component implement a massive parallel computation or alternatively observe where new computations are really needed and achieve these computations.

Execution model: We use the massive parallel model with a locally parallel, globally sequential approach. The data space is divided in stripes recorded in

node memories. The width of the stripes is the bandwidth to the FPGAs (32 bits × number of nodes). One or two nodes are in charge of feeding data dependencies on the slice borders and to read back these dependencies for the future step.

Processors manage two spaces for current state and next state. Their activity is to repetitively transform their current stripe into a new one. They need to exchange values because of the dependencies on the stripe borders.

Program expression: Programs are expressed in a simple syntax covering the three definitions given previously:

1. each cell state is described as a C record grouping bit-field variables,
2. the neighborhood is declared as a set of directions (C, for center, N for north, NW for north-west...),
3. the data space is declared by two integer values for with and height,
4. the transition function is a C function returning the new value of the local cell computed from the neighborhood state.

Architecture description: As CAs can be considered as fine grain computations involving a lot of data exchanges, the transition function will be synthesized in hardware [7]. To enable this function to proceed, it is needed to present the neighborhood. Thus, a simple approach for the architecture is to provide a FIFO in which the cells are progressing, and to connect this FIFO to a row of processors. Dependencies can be wired between adjacent nodes. Control is the responsibility of the processors that permanently read their memories, write to the FPGAs, read back the new state from the FPGA to write it to memory. Their coordination is enforced during the accesses by local handshakes. Several time steps can be cascaded along a pipeline.

Physical synthesis: Architecture implementation is a fully automatic process leaded by a dedicated synthesizer. The availability of tools described in section 1 allows a constructive approach of the physical synthesis. The constraints that need to be observed are the cell width, the data path width in the FPGA, the size of the processors, the possible saturation of routing resources. Physical layout can be achieved using Madeo tools, following these steps: (1) synthesize, (2) place and route the processor, (3) compute the FIFO size, (4) make an estimation of the routing channel width, (5) place the processors on the FPGA dye, place the FIFO registers, (6) call the point to point router to connect registers together, (7) connect the registers to the processor, connect the processor to the feed back lines, connect the registers to the interface, (8) connect the clock to the interface.

Stacking components: Models can be stacked. As an example, we have produced a partial implementation of the Wu and Manber pattern matching algorithm[8, 9] above the CA component. In this case, the program becomes a pattern to be searched and the number of errors that are accepted. Implementation of some low level operators for image processing is also immediate.

Physical and computational constraints: Physical synthesis for a component is a determinist approach dealing on one side with the reconfigurable unit

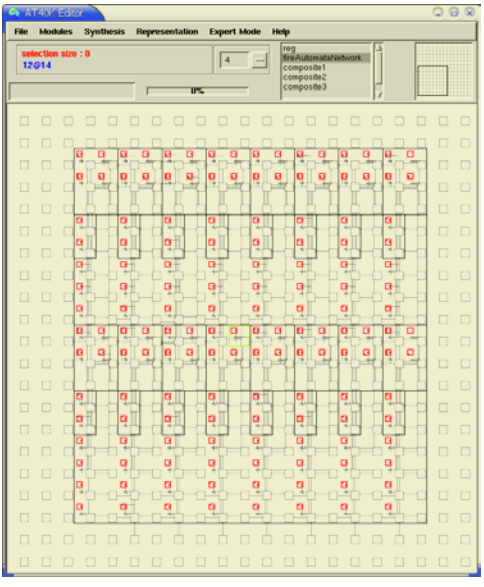

Fig. 3. Layout of a cellular automata for fire propagation on an Atmel 40K FPGA. The circuit has small processors which internal state represent the situation on 2 bits at a geographic position. Two stages of automata had been cascaded, and 8 slices are represented.

organization and resources, and on the other side with a characterization of the component. The behavior of the component is fixed by a high level program block processed by the logic synthesizer. Another important issue is the internal development of parallelism during synthesis. Care must be taken not to waste hardware resources by adapting synthesis algorithms to the usable data rate on the unit interface.

4 Physical Synthesis: Systolic Array Example

Given a specific applicative model, this part shows what are the advantages of a specific component definition and algorithms for physical synthesis.

Characterizing a systolic array: Systolic arrays (SA) represent intensive computations as found in nested loops, in various applicative domains (digital signal processing, DNA comparison, image processing, ...). Basically the corresponding architecture is a regular array of processing elements (PE) performing efficiently the body of the inner loop. All PEs are only connected to their neighbors except for the first and last PEs that are connected to an outside system. Systolic arrays are one of the more structured and regular component and are described by a few number of characteristics (figure 4.a): (1) the inputs, outputs and functionality of one PE, (2) the interconnection pattern between two neigh-

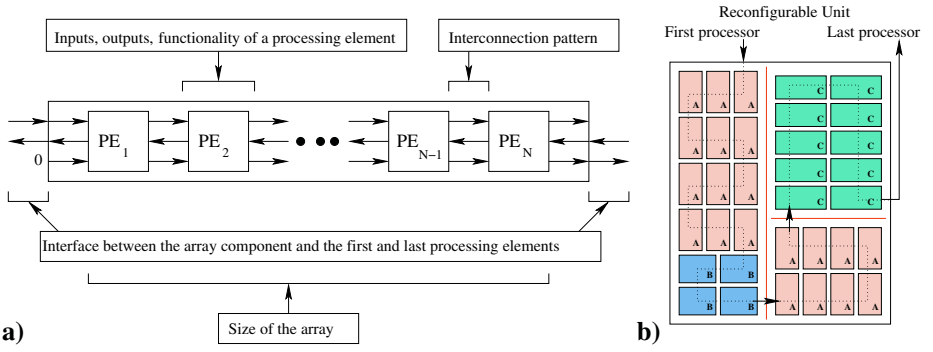

Fig. 4. (a) Characterizing a systolic array. (b) Placing a systolic array on a reconfigurable unit using structural properties.

bor PEs, (3) the interface between I/O of the Systolic Array and I/O of the first and last processors, (4) the size of the array.

Using structural properties in the design flow: Regularity in a SA occurs in the PE and the interconnection pattern. As all PEs have an identical structure, an implementation pattern for a PE structure can be replicated for all PEs. Depending on limitations related to design tools and target technology, productivity gains could occur in each step of the design flow :

1. *Synthesis, optimizations and mapping:* whether the processor description is behavioral or structural, synthesis, optimization and mapping are just operating on one processor bounding box, reducing drastically the complexity of this step.
2. *Placing:* as for the previous step, complexity is reduced to placing one PE, then simultaneously replicating and floorplanning it for the whole array. Floorplanning complexity is lower than placing a flat design, since it acts on coarser grain component, and that we have the capabilities to constraint the placement of one PE to a geometric shape easier to floorplan. Moreover the floorplanning is automatically deduced from the systolic array topologies (1D or 2D grid).
3. *Routing:* Replicating the routing scheme of a PE and the routing pattern between two neighbor PEs is possible if routing conflicts are overcome.

This design flow induces lot of savings related to classical flow because it takes into account SA properties. All these optimizations decrease the design runtime and increase the productivity. Moreover, by mapping physically the systolic array structure to a reconfigurable unit, savings also include increased clock frequency (by reducing wire length).

A method to use structural properties for placement: As an example, we present results of the FRAP tool [10], which aims to put the maximum number of PEs of a linear SA on a reconfigurable unit, given various constraints, by adding

placement directives to a structural SA description. Finding such a placement acts in three steps (see figure 4.b for a basic example):

(1) All possible geometric shapes for a PE are generated by combining all shapes of its sub-components, (2) a full snake-like placement is determined using the processing element shapes previously computed, (3) the final internal placement of the processing elements is performed according to their shapes.

Experiments show several savings resulting from the use of this tool. Placement step runtime is divided up to factor of 6. Routing step runtime is divided up to factor of 3. Clock frequency is increased up to a factor of 2. However in some cases results are quite limited and even worst, principally due to lack of control over the vendor design tools that were used.

Why and how to take care of design structure and regularity? Given the previous example of using structural properties for physical synthesis, we can extent this principle to all structured components. Once described in a HLL, a circuit structure is deduced from DAG. The structure of a circuit will have different degrees of regularity, occurring at various hierarchical levels, ranging from low (identical slices of an adder) to high (identical PEs of a regular array). It is even possible to extract structure and regularity from a flat design. A distinction must be made between the advantages just induced by knowing the structure, and the advantages induced by having regularity in the structure:

1. *Structure* keeps information about the interconnections (logical optimization, mapping, placement) without needing to recompute it at each design flow step. It principally permits to improve design density and frequency.
2. *Regularity* permits to reduce the design flow runtime, by merging tasks, meaning to find a solution for a structural template and replicate it for all entities assimilated to this template. This method is applied recursively over the component hierarchy, being applied in the steps of synthesis, logical optimization, mapping, placement and routing, if the software or technological environment permits it.

From those two criterion, taking care of design structure increases density, frequency and decreases design flow running time. As applicative design and reconfigurable unit area become larger, using structural properties will allow to deal with the increasing complexity of physical synthesis, although by the past this approach offered limited improvement and big effort to develop specific tools, due to the need to be adapted to closed vendor design environment.

5 Conclusion

In the context of reconfigurable heterogeneous platforms, we are proposing a method allowing to produce components from productive development tools. These components can be synthesized for different compute units such as processors or fine grain FPGAs. We are actively working to rise the capabilities of synthesis tools to address mixed grain units and implement transformations such as loop unrolling.

This approach has been made possible by the development of the Madeo open framework in which target reconfigurable architectures can be represented, with the immediate feed-back on basic tools for physical design. The cellular automata example that was actually implemented and executed on the ArMen computer, is now drawn on recent FPGAs without any black-box "support". The new tools created in the object-oriented environment are considerably easing developments, portability, modular assembly of components. The case of systolic arrays is significant in terms of physical design problems since these circuits can be described with simplicity, they produce a lot of computing power, and they are resource hungry.

While there is no reason to restrict the component design method to object-oriented languages and tools, it is expected that such environments will ease the management of run-time exchanges, application development and system activity description.

Acknowledgements

Parts of this work are supported by the MEFI/STSI and by the Ministry of Research *RNTL*. Thanks are due to T. Ben-Ismail and the AST division of STMicroelectronics for their support and cooperation.

References

1. Lagadec, L.: Abstraction, modélisation et outils de CAO pour les circuits intégrés reconfigurables. PhD thesis, Université de Rennes 1 (2000)
2. Cong, J., Ding, Y.: Combinational logic synthesis for lut based fpga. ACM transaction on DAES (1996)
3. Lagadec, L., Pottier, B., Villellas-Guillen, O.: An lut-based high level synthesis framework for reconfigurable architectures. In: Domain-Specific Processors : Systems, Architectures, Modeling, and Simulation, Marcel Dekker (2003)
4. Nicoud, J.D., Tyrrell, A.M.: The transputer t414 instruction set. IEEE Micro **9** (1989)
5. Spinellis, D.: Reliable software implementation using domain specific languages. In: ESREL, 10th european conference on safety and reliability. (1999)
6. Dhaussy, P., Filloque, J.M., Pottier, B., Rubini, S.: Global Control Synthesis for an MIMD/FPGA Machine. In: FPGAs for Custom Computing Machines. (1994)
7. Bouazza, K., Champeau, J., Ng, P., Pottier, B., Rubini, S.: Implementing cellular automata on the ArMen machine. In: Algorithms and Parallel VLSI Architectures II, Elseiver (1991)
8. Wu, S., Manber, U.: Fast text searching allowing errors. Communications of the ACM **35** (1992)
9. Champeau, J., Le Pape, L., Pottier, B.: Parallel Grep. In: Algorithms and Parallel VLSI Architectures III, Leuven, Belgium, Elsevier (1994)
10. Fabiani, E., Lavenier, D.: Experimental evaluation of place-and-route of regular arrays on xilinx chips. In: First International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, USA (2001)